Working with data

17

This chapter covers

- Caching data values or complete responses using ASP.NET Core
- Working with Entity Framework Core to access data in a relational database

All the examples in the earlier chapters in this part of the book have generated fresh responses for each request, which is easy to do when dealing with simple strings or small fragments of HTML. Most real projects deal with data that is expensive to produce and needs to be used as efficiently as possible. In this chapter, I describe the features that ASP.NET Core provides for caching data and caching entire responses. I also show you how to create and configure the services required to access data in a database using Entity Framework Core. Table 17.1 puts the ASP.NET Core features for working with data in context.

> **NOTE** The examples in this chapter rely on the SQL Server LocalDB feature that was installed in chapter 2. You will encounter errors if you have not installed LocalDB and the required updates.

Table 17.1 Putting the ASP.NET Core data features in context

Question	Answer
What are they?	The features described in this chapter allow responses to be produced using data that has been previously created, either because it was created for an earlier request or because it has been stored in a database.
Why are they useful?	Most web applications deal with data that is expensive to re-create for every request. The features in this chapter allow responses to be produced more efficiently and with fewer resources.
How are they used?	Data values are cached using a service. Responses are cached by a middleware component based on the `Cache-Control` header. Databases are accessed through a service that translates LINQ queries into SQL statements.
Are there any pitfalls or limitations?	For caching, it is important to test the effect of your cache policy before deploying the application to ensure you have found the right balance between efficiency and responsiveness. For Entity Framework Core, it is important to pay attention to the queries sent to the database to ensure that they are not retrieving large amounts of data that is processed and then discarded by the application.
Are there any alternatives?	All the features described in this chapter are optional. You can elect not to cache data or responses or to use an external cache. You can choose not to use a database or to access a database using a framework other than Entity Framework Core.

Table 17.2 provides a guide to the chapter.

Table 17.2 Chapter guide

Problem	Solution	Listing
Caching data values	Set up a cache service and use it in endpoints and middleware components to store data values.	3–6
Creating a persistent cache	Use the database-backed cache.	7–13
Caching entire responses	Enable the response or output caching middleware.	14-19
Storing application data	Use Entity Framework Core.	20-26, 29-31
Creating a database schema	Create and apply migrations.	27, 28
Accessing data in endpoints	Consume the database context service.	32
Including all request details in logging messages	Enable the sensitive data logging feature.	33

17.1   *Preparing for this chapter*

In this chapter, I continue to use the `Platform` project from chapter 16. To prepare for this chapter, replace the contents of the `Program.cs` file with the code shown in listing 17.1.

> **TIP**   You can download the example project for this chapter—and for all the other chapters in this book—from https://github.com/manningbooks/pro-asp .net-core-7. See chapter 1 for how to get help if you have problems running the examples.

Listing 17.1   Replacing the contents of the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/", async context => {
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

Start the application by opening a new PowerShell command prompt, navigating to the `Platform` project folder (which contains the `Platform.csproj` file), and running the command shown in listing 17.2.

Listing 17.2   Starting the ASP.NET Core runtime

```
dotnet run
```

Open a new browser tab, navigate to https://localhost:5000, and you will see the content shown in figure 17.1.
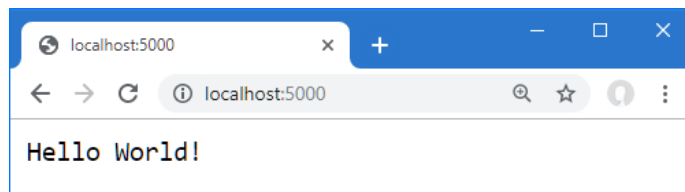

Figure 17.1   Running the example application

17.2   *Caching data*

In most web applications, there will be some items of data that are relatively expensive to generate but are required repeatedly. The exact nature of the data is specific to each project, but repeatedly performing the same set of calculations can increase the resources required to host the application. To represent an expensive response, add a class file called `SumEndpoint.cs` to the `Platform` folder with the code shown in listing 17.3.

> **Listing 17.3 The contents of the SumEndpoint.cs file in the Platform folder**

```
namespace Platform {

    public class SumEndpoint {

        public async Task Endpoint(HttpContext context) {
            int count;
            int.TryParse((string?)context.Request.RouteValues["count"],
                out count);
            long total = 0;
            for (int i = 1; i <= count; i++) {
                total += i;
            }
            string totalString = $"({DateTime.Now.ToLongTimeString()}) "
                + total;
            await context.Response.WriteAsync(
                $"({DateTime.Now.ToLongTimeString()}) Total for {count}"
                + $" values:\n{totalString}\n");
        }
    }
}
```

Listing 17.4 creates a route that uses the endpoint, which is applied using the Map-Endpoint extension methods created in chapter 14.

> **Listing 17.4 Adding an endpoint in the Program.cs file in the Platform folder**

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapEndpoint<Platform.SumEndpoint>("/sum/{count:int=1000000000}");

app.MapGet("/", async context => {
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

Restart ASP.NET Core and use a browser to request http://localhost:5000/sum. The endpoint will sum 1,000,000,000 integer values and produce the result shown in figure 17.2.

Reload the browser window, and the endpoint will repeat the calculation. Both the timestamps change in the response, as shown in the figure, indicating that every part of the response was produced fresh for each request.

> **TIP** You may need to increase or decrease the default value for the route param-eter based on the capabilities of your machine. Try to find a value that takes two or three seconds to produce the result—just long enough that you can tell when the calculation is being performed but not so long that you can step out for cof-fee while it happens.
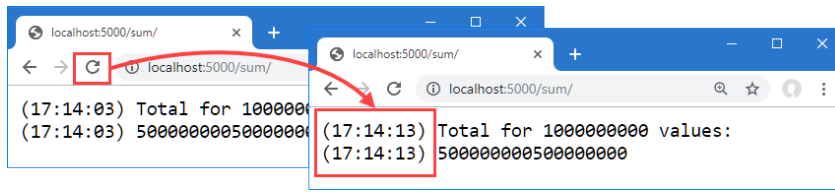
Figure 17.2   An expensive response

17.2.1 *Caching data values*

ASP.NET Core provides a service that can be used to cache data values through the IDistributedCache interface. Listing 17.5 revises the endpoint to declare a dependency on the service and use it to cache calculated values.

Listing 17.5   Using the cache service in the SumEndpoint.cs file in the Platform folder

```
using Microsoft.Extensions.Caching.Distributed;

namespace Platform {

    public class SumEndpoint {

        public async Task Endpoint(HttpContext context,
                IDistributedCache cache) {
            int count;
            int.TryParse((string?)context.Request.RouteValues["count"],
                out count);
            string cacheKey = $"sum_{count}";
            string? totalString = await cache.GetStringAsync(cacheKey);
            if (totalString == null) {
                long total = 0;
                for (int i = 1; i <= count; i++) {
                    total += i;
                }
                totalString = $"({DateTime.Now.ToLongTimeString()}) "
                    + total;
                await cache.SetStringAsync(cacheKey, totalString,
                    new DistributedCacheEntryOptions {
                        AbsoluteExpirationRelativeToNow =
                            TimeSpan.FromMinutes(2)
                    });
            }
            await context.Response.WriteAsync(
                $"({DateTime.Now.ToLongTimeString()}) Total for {count}"
                + $" values:\n{totalString}\n");
        }
    }
}
```

The cache service can store only byte arrays, which can be restrictive but allows for a range of IDistributedCache implementations to be used. There are extension

methods available that allow strings to be used, which is a more convenient way of caching most data. Table 17.3 describes the most useful methods for using the cache.

Table 17.3   Useful IDistributedCache methods

Name	Description
`GetString(key)`	This method returns the cached string associated with the specified key, or `null` if there is no such item.
`GetStringAsync(key)`	This method returns a `Task<string>` that produces the cached string associated with the key, or `null` if there is no such item.
`SetString(key, value, options)`	This method stores a string in the cache using the specified key. The cache entry can be configured with an optional `DistributedCacheEntryOptions` object.
`SetStringAsync(key, value, options)`	This method asynchronously stores a string in the cache using the specified key. The cache entry can be configured with an optional `Distributed-CacheEntryOptions` object.
`Refresh(key)`	This method resets the expiry interval for the value associated with the key, preventing it from being flushed from the cache.
`RefreshAsync(key)`	This method asynchronously resets the expiry interval for the value associated with the key, preventing it from being flushed from the cache.
`Remove(key)`	This method removes the cached item associated with the key.
`RemoveAsync(key)`	This method asynchronously removes the cached item associated with the key.

By default, entries remain in the cache indefinitely, but the `SetString` and `SetString-Async` methods accept an optional `DistributedCacheEntryOptions` argument that is used to set an expiry policy, which tells the cache when to eject the item. Table 17.4 shows the properties defined by the `DistributedCacheEntryOptions` class.

Table 17.4   The DistributedCacheEntryOptions properties

Name	Description
`AbsoluteExpiration`	This property is used to specify an absolute expiry date.
`AbsoluteExpirationRelativeToNow`	This property is used to specify a relative expiry date.
`SlidingExpiration`	This property is used to specify a period of inactivity, after which the item will be ejected from the cache if it hasn't been read.

In listing 17.5, the endpoint uses the `GetStringAsync` to see whether there is a cached result available from a previous request. If there is no cached value, the endpoint performs the calculation and caches the result using the `SetStringAsync` method, with the `AbsoluteExpirationRelativeToNow` property to tell the cache to eject the item after two minutes.

```
...
await cache.SetStringAsync(cacheKey, totalStr,
    new DistributedCacheEntryOptions {
        AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(2)
    });
...
```

The next step is to set up the cache service, as shown in listing 17.6.

> **Listing 17.6   Adding a service in the Program.cs file in the Platform folder**

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDistributedMemoryCache(opts => {
    opts.SizeLimit = 200;
});

var app = builder.Build();

app.MapEndpoint<Platform.SumEndpoint>("/sum/{count:int=1000000000}");

app.MapGet("/", async context => {
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

`AddDistributedMemoryCache` is the same method I used in chapter 16 to provide the data store for session data. This is one of the three methods used to select an implementation for the `IDistributedCache` service, as described in table 17.5.

Table 17.5   The cache service implementation methods

Name	Description
AddDistributedMemoryCache	This method sets up an in-memory cache.
AddDistributedSqlServerCache	This method sets up a cache that stores data in SQL Server and is available when the `Microsoft.Extensions.Caching.SqlServer` package is installed. See the "Caching Responses" section for details.
AddStackExchangeRedisCache	This method sets up a Redis cache and is available when the `Microsoft.Extensions.Caching.Redis` package is installed.

Listing 17.6 uses the `AddDistributedMemoryCache` method to create an in-memory cache as the implementation for the `IDistributedCache` service. This cache is

configured using the `MemoryDistributedCacheOptions` class, whose most useful properties are described in table 17.6.

Table 17.6   Useful MemoryCacheOptions properties

Name	Description
`ExpirationScanFrequency`	This property is used to set a `TimeSpan` that determines how often the cache scans for expired items.
`SizeLimit`	This property specifies the maximum number of items in the cache. When the size is reached, the cache will eject items.
`CompactionPercentage`	This property specifies the percentage by which the size of the cache is reduced when `SizeLimit` is reached.

The statement in listing 17.6 uses the `SizeLimit` property to restrict the cache to 200 items. Care must be taken when using an in-memory cache to find the right balance between allocating enough memory for the cache to be effective without exhausting server resources.

To see the effect of the cache, restart ASP.NET Core and request the http:// localhost:5000/sum URL. Reload the browser, and you will see that only one of the timestamps will change, as shown in figure 17.3. This is because the cache has provided the calculation response, which allows the endpoint to produce the result without having to repeat the calculation.
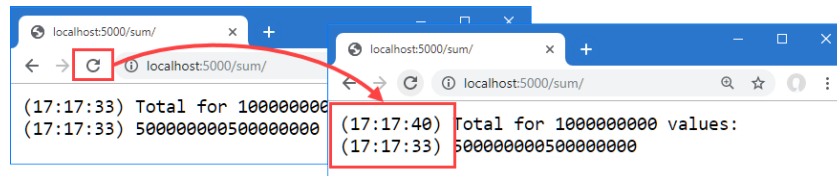
Figure 17.3.   Caching data values

If you wait for two minutes and then reload the browser, then both timestamps will change because the cached result will have been ejected, and the endpoint will have to perform the calculation to produce the result.

17.2.2 *Using a shared and persistent data cache*

The cache created by the `AddDistributedMemoryCache` method isn't distributed, despite the name. The items are stored in memory as part of the ASP.NET Core process, which means that applications that run on multiple servers or containers don't share cached data. It also means that the contents of the cache are lost when ASP.NET Core is stopped.

The `AddDistributedSqlServerCache` method stores the cache data in a SQL Server database, which can be shared between multiple ASP.NET Core servers and which stores the data persistently.

The first step is to create a database that will be used to store the cached data. You can store the cached data alongside the application's other data, but for this chapter, I am going to use a separate database, which will be named `CacheDb`. You can create the database using Azure Data Studio or SQL Server Management Studio, both of which are available for free from Microsoft. Databases can also be created from the command line using `sqlcmd`. Open a new PowerShell command prompt and run the command shown in listing 17.7 to connect to the LocalDB server.

> **TIP**   The `sqlcmd` tool should have been installed as part of the Visual Studio workload or as part of the SQL Server Express installation. If it has not been installed, then you can download an installer from https://docs.microsoft.com/en-us/sql/tools/sqlcmd-utility.

> **Listing 17.7   Connecting to the database server**

```
sqlcmd -S "(localdb)\MSSQLLocalDB"
```

Pay close attention to the argument that specifies the database. There is one backslash, which is followed by `MSSQLLocalDB`. It can be hard to spot the repeated letters: M**S**-**SQL**-**L**ocalDB (but without the hyphens).

When the connection has been established, you will see a `1>` prompt. Enter the commands shown in listing 17.8 and press the Enter key after each command.

> **CAUTION**   If you are using Visual Studio, you must apply the updates for SQL Server described in chapter 2. The version of SQL Server that is installed by default when you install Visual Studio cannot create LocalDB databases.

> **Listing 17.8   Creating the database**

```
CREATE DATABASE CacheDb
GO
```

If no errors are reported, then enter `exit` and press Enter to terminate the connection.

> **TIP**   If you need to reset the cache database, use the command in listing 17.7 to open a connection and use the command `DROP DATABASE CacheDB`. You can then re-create the database using the commands in listing 17.8.

Run the commands shown in listing 17.9 to install the package required to create a cache.

> **Listing 17.9   Installing the SQL cache package**

```
dotnet tool uninstall --global dotnet-sql-cache
dotnet tool install --global dotnet-sql-cache --version 7.0.0
```

The first command removes any existing version of the `dotnet-sql-cache` package, and the second command installs the version required for the examples in this book. The next step is to run the command shown in listing 17.10 to create a table in the new database, using the command installed by the `dotnet-sql-cache` package.

Listing 17.10 Creating the cache database table

```
dotnet sql-cache create "Server=(localdb)\MSSQLLocalDB;Database=CacheDb"
dbo DataCache
```

The arguments for this command are the connection string that specifies the database, the schema, and the name of the table that will be used to store the cached data. Enter the command on a single line and press Enter. It will take a few seconds for the tool to connect to the database. If the process is successful, you will see the following message:

```
Table and index were created successfully.
```

CREATING THE PERSISTENT CACHE SERVICE

Now that the database is ready, I can create the service that will use it to store cached data. To add the NuGet package required for SQL Server caching support, open a new PowerShell command prompt, navigate to the `Platform` project folder, and run the command shown in listing 17.11. (If you are using Visual Studio, you can add the package by selecting Project > Manage NuGet Packages.)

Listing 17.11 Adding a package to the project

```
dotnet add package Microsoft.Extensions.Caching.SqlServer --version 7.0.0
```

The next step is to define a connection string, which describes the database connection in the JSON configuration file, as shown in listing 17.12.

> **NOTE** The cache created by the `AddDistributedSqlServerCache` method is distributed, meaning that multiple applications can use the same database and share cache data. If you are deploying the same application to multiple servers or containers, all instances will be able to share cached data. If you are sharing a cache between different applications, then you should pay close attention to the keys you use to ensure that applications receive the data types they expect.

Listing 17.12 A connection string in the appsettings.json file in the Platform folder

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Location": {
    "CityName": "Buffalo"
  },
```

```
  "ConnectionStrings": {
    "CacheConnection": "Server=(localdb)\\MSSQLLocalDB;Database=CacheDb"
  }
}
```

Notice that the connection string uses two backslash characters (\\) to escape the character in the JSON file. Listing 17.13 changes the implementation for the cache service to use SQL Server with the connection string from listing 17.12.

> **Listing 17.13   Using a persistent data cache in the Program.cs file in the Platform folder**

```
var builder = WebApplication.CreateBuilder(args);

//builder.Services.AddDistributedMemoryCache(opts => {
//    opts.SizeLimit = 200;
//});

builder.Services.AddDistributedSqlServerCache(opts => {
    opts.ConnectionString
        = builder.Configuration["ConnectionStrings:CacheConnection"];
    opts.SchemaName = "dbo";
    opts.TableName = "DataCache";
});

var app = builder.Build();

app.MapEndpoint<Platform.SumEndpoint>("/sum/{count:int=1000000000}");

app.MapGet("/", async context => {
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

The `IConfiguration` service is used to access the connection string from the application's configuration data. The cache service is created using the `Add-DistributedSqlServerCache` method and is configured using an instance of the `SqlServerCacheOptions` class, whose most useful properties are described in table 17.7.

Table 17.7.   Useful SqlServerCacheOptions properties

Name	Description
ConnectionString	This property specifies the connection string, which is conventionally stored in the JSON configuration file and accessed through the `IConfiguration` service.
SchemaName	This property specifies the schema name for the cache table.
TableName	This property specifies the name of the cache table.
ExpiredItemsDeletionInterval	This property specifies how often the table is scanned for expired items. The default is 30 minutes.
DefaultSlidingExpiration	This property specifies how long an item remains unread in the cache before it expires. The default is 20 minutes.

The listing uses the `ConnectionString`, `SchemaName`, and `TableName` properties to configure the cache middleware to use the database table. Restart ASP.NET Core and use a browser to request the http://localhost:5000/sum URL. There is no change in the response produced by the application, which is shown in figure 17.4, but you will find that the cached responses are persistent and will be used even when you restart ASP.NET Core.

> ### Caching session-specific data values
>
> When you use the `IDistributedCache` service, the data values are shared between all requests. If you want to cache different data values for each user, then you can use the session middleware described in chapter 16. The session middleware relies on the `IDistributedCache` service to store its data, which means that session data will be stored persistently and available to a distributed application when the `Add-DistributedSqlServerCache` method is used.

17.3 Caching responses

An alternative to caching individual data items is to cache entire responses, which can be a useful approach if a response is expensive to compose and is likely to be repeated. Caching responses requires the addition of a service and a middleware component, as shown in listing 17.14.

> **Listing 17.14   Configuring Caching in the Program.cs File in the Platform Folder**

```
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDistributedSqlServerCache(opts => {
    opts.ConnectionString
        = builder.Configuration["ConnectionStrings:CacheConnection"];
    opts.SchemaName = "dbo";
    opts.TableName = "DataCache";
});

builder.Services.AddResponseCaching();
builder.Services.AddSingleton<IResponseFormatter,
    HtmlResponseFormatter>();

var app = builder.Build();

app.UseResponseCaching();

app.MapEndpoint<Platform.SumEndpoint>("/sum/{count:int=1000000000}");

app.MapGet("/", async context => {
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

The `AddResponseCaching` method is used to set up the service used by the cache. The middleware component is added with the `UseResponseCaching` method, which should be called before any endpoint or middleware that needs its responses cached.

I have also defined the `IResponseFormatter` service, which I used to explain how dependency injection works in chapter 14. Response caching is used only in certain circumstances, and, as I explain shortly, demonstrating the feature requires an HTML response.

> **NOTE**   The response caching feature does not use the `IDistributedCache` service. Responses are cached in memory and are not distributed.

In listing 17.15, I have updated the `SumEndpoint` class so that it requests response caching instead of caching just a data value.

> **Listing 17.15   Using response caching in the SumEndpoint.cs file in the Platform folder**

```
//using Microsoft.Extensions.Caching.Distributed;
using Platform.Services;

namespace Platform {

    public class SumEndpoint {

        public async Task Endpoint(HttpContext context,
                IResponseFormatter formatter, LinkGenerator generator) {

            int count;
            int.TryParse((string?)context.Request.RouteValues["count"],
                out count);
            long total = 0;
            for (int i = 1; i <= count; i++) {
                total += i;
            }
            string totalString = $"({DateTime.Now.ToLongTimeString()}) "
                + total;

            context.Response.Headers["Cache-Control"]
                = "public, max-age=120";

            string? url = generator.GetPathByRouteValues(context, null,
                new { count = count });

            await formatter.Format(context,
                $"<div>({DateTime.Now.ToLongTimeString()}) Total for "
                + $"{count} values:</div><div>{totalString}</div>"
                + $"<a href={url}>Reload</a>");
        }
    }
}
```

Some of the changes to the endpoint enable response caching, but others are just to demonstrate that it is working. For enabling response caching, the important statement is the one that adds a header to the response, like this:

```
...
context.Response.Headers["Cache-Control"] = "public, max-age=120";
...
```

The `Cache-Control` header is used to control response caching. The middleware will only cache responses that have a `Cache-Control` header that contains the `public` directive. The `max-age` directive is used to specify the period that the response can be cached for, expressed in seconds. The `Cache-Control` header used in listing 17.15 enables caching and specifies that responses can be cached for two minutes.

Enabling response caching is simple, but checking that it is working requires care. When you reload the browser window or press Return in the URL bar, browsers will include a `Cache-Control` header in the request that sets the `max-age` directive to zero, which bypasses the response cache and causes a new response to be generated by the endpoint. The only reliable way to request a URL without the `Cache-Control` header is to navigate using an HTML anchor element, which is why the endpoint in listing 17.15 uses the `IResponseFormatter` service to generate an HTML response and uses the `LinkGenerator` service to create a URL that can be used in the anchor element's `href` attribute.

To check the response cache, restart ASP.NET Core and use the browser to request http://localhost:5000/sum. Once the response has been generated, click the Reload link to request the same URL. You will see that neither of the timestamps in the response change, indicating that the entire response has been cached, as shown in figure 17.4.
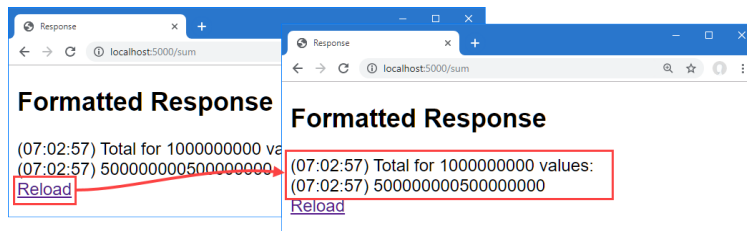
Figure 17.4
Caching
responses

The `Cache-Control` header can be combined with the `Vary` header to provide fine-grained control over which requests are cached. See https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control and https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Vary for details of the features provided by both headers.

Compressing responses

ASP.NET Core includes middleware that will compress responses for browsers that have indicated they can handle compressed data. The middleware is added to the pipeline with the `UseResponseCompression` method. Compression is a trade-off between the server resources required for compression and the bandwidth required to deliver content to the client, and it should not be switched on without testing to determine the performance impact.

17.4 *Caching output*

The use of the `Cache-Control` header follows the intended design of HTTP but limits the benefits of caching responses. ASP.NET Core includes an alternative approach to caching, known as *output caching*, which overcomes these problems and provides a more configurable—albeit complex—set of features. Caching is applied to endpoints and to prepare for this example, I need to update the code written in chapter 14 so that it produces a result on which the caching extension method can be invoked, as shown in listing 17.16.

> **NOTE**   The output cache feature is more comprehensive than response caching but has its own limitations. There is an API for implementing persistent cache storage, for example, but no default implementation. The distributed cache used in earlier examples can't be used because output caching requires careful locking of cached data.

Listing 17.16   A result in the EndpointExtensions.cs file in the Platform/Services folder

```
using System.Reflection;

namespace Microsoft.AspNetCore.Builder {

    public static class EndpointExtensions {

        public static IEndpointConventionBuilder MapEndpoint<T>(
                this IEndpointRouteBuilder app,
                string path, string methodName = "Endpoint") {

            MethodInfo? methodInfo = typeof(T).GetMethod(methodName);
            if (methodInfo?.ReturnType != typeof(Task)) {
                throw new System.Exception("Method cannot be used");
            }

            ParameterInfo[] methodParams = methodInfo!.GetParameters();

            return app.MapGet(path, context => {
                T endpointInstance =
                    ActivatorUtilities.CreateInstance<T>
                        (context.RequestServices);
                return (Task)methodInfo.Invoke(endpointInstance!,
                    methodParams.Select(p =>
                    p.ParameterType == typeof(HttpContext)
                    ? context
                    : context.RequestServices.GetService(p.ParameterType))
                        .ToArray())!;
            });
        }
    }
}
```

This change alters the `MapEndpoint<T>` extension method to produce an implementation of the `IEndpointConventionBuilder` interface, which is the result produced by

the built-in `MapGet` method and the other methods used to create endpoints. Listing 17.17 replaces the response caching from the previous section with output caching.

> **Listing 17.17   Using output caching in the Program.cs file in the Platform folder**

```
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

//builder.Services.AddDistributedSqlServerCache(opts => {
//    opts.ConnectionString
//        = builder.Configuration["ConnectionStrings:CacheConnection"];
//    opts.SchemaName = "dbo";
//    opts.TableName = "DataCache";
//});

builder.Services.AddOutputCache();

//builder.Services.AddResponseCaching();
builder.Services.AddSingleton<IResponseFormatter,
    HtmlResponseFormatter>();

var app = builder.Build();

//app.UseResponseCaching();
app.UseOutputCache();

app.MapEndpoint<Platform
    .SumEndpoint>("/sum/{count:int=1000000000}")
    .CacheOutput();

app.MapGet("/", async context => {
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

This is the simplest output caching configuration, and it is applied in three parts. First, the `AddOutputCache` method is called to set up the services required for caching. Second, the `UseOutputCache` method is called to register the caching middleware, which will short-circuit the pipeline where requests are received that can be handled using cached content. The final step enables caching for a specific endpoint, which is done using the `CacheOutput` extension method:

```
...
app.MapEndpoint<Platform
    .SumEndpoint>("/sum/{count:int=1000000000}")
    .CacheOutput();
...
```

This extension method enables caching for the sum endpoint. The final step is to update the endpoint so that it doesn't set the caching header, as shown in listing 17.18.

> **Listing 17.18   Disabling the header in the SumEndpoint.cs file in the Platform folder**

```
//using Microsoft.Extensions.Caching.Distributed;
using Platform.Services;

namespace Platform {

    public class SumEndpoint {

        public async Task Endpoint(HttpContext context,
                IResponseFormatter formatter, LinkGenerator generator) {

            int count;
            int.TryParse((string?)context.Request.RouteValues["count"],
                out count);
            long total = 0;
            for (int i = 1; i <= count; i++) {
                total += i;
            }
            string totalString = $"({DateTime.Now.ToLongTimeString()}) "
                + total;

            //context.Response.Headers["Cache-Control"]
            //    = "public, max-age=120";

            string? url = generator.GetPathByRouteValues(context, null,
                new { count = count });

            await formatter.Format(context,
                $"<div>({DateTime.Now.ToLongTimeString()}) Total for "
                + $"{count} values:</div><div>{totalString}</div>"
                + $"<a href={url}>Reload</a>");
        }
    }
}
```

The endpoint doesn't need to manage the caching of its content directly, which means that the cache settings can be altered without needing to change the code that generates responses.

The configuration used in listing 17.18 applies the default caching policy, which caches content for one minute and caches only HTTP GET or HEAD requests that produce HTTP 200 responses, and which are not authenticated or set cookies.

To check the output cache, restart ASP.NET Core and use the browser to request http://localhost:5000/sum. You will see the same output as in earlier examples, but now the cache is applied when the browser is reloaded and not just when the user clicks on the link, as shown in figure 17.5.
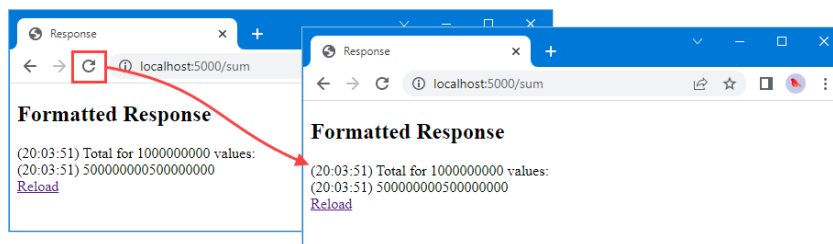
Figure 17.5. Output caching applies to all requests for a given URL

17.4.1 *Defining a custom cache policy*

The output caching feature supports custom caching policies and allows different poli-
cies to be applied to endpoints. Caching policies are defined using the options pattern,
as shown in listing 17.19.

Listing 17.19   A custom configuration in the Program.cs file in the Platform folder

```
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddOutputCache(opts => {
    opts.AddBasePolicy(policy => {
        policy.Cache();
        policy.Expire(TimeSpan.FromSeconds(10));
    });
    opts.AddPolicy("30sec", policy => {
        policy.Cache();
        policy.Expire(TimeSpan.FromSeconds(30));
    });
});

builder.Services.AddSingleton<IResponseFormatter,
    HtmlResponseFormatter>();

var app = builder.Build();

app.UseOutputCache();

app.MapEndpoint<Platform
    .SumEndpoint>("/sum/{count:int=1000000000}")
    .CacheOutput();

app.MapEndpoint<Platform
    .SumEndpoint>("/sum30/{count:int=1000000000}")
    .CacheOutput("30sec");

app.MapGet("/", async context => {
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

The options pattern is applied to an `OutputCacheOptions` object, whose most useful methods are described in table 17.8.

Table 17.8 Useful OutputCacheOptions methods

Name	Description
AddBasePolicy	This method is used to define the default policy
AddPolicy	This method is used to create a policy that can be applied to specific endpoints.

The methods described in table 17.8 define policies using the methods defined by the `OutputCachePolicyBuilder` class. The basic methods are described in table 17.9.

Table 17.9 The basic OutputCachePolicyBuilder methods

Name	Description
Expire	This method sets the expiry interval for cached output, expressed as a `TimeSpan`.
SetLocking	This method is used to enable or disable locking in the cache. Locking is enabled by default and should only be disabled with caution since it can lead to unexpected behavior.
NoCache	This method disables caching in the policy.
Cache	This method enables caching in the policy.

The policies in listing 17.19 use the `Cache` and `Expire` methods. The `AddBasePolicy` method is used to change the cache duration for the default policy. The `AddPolicy` method is used to create a new cache policy named `30sec`, which caches output for 30 seconds:

```
...
opts.AddPolicy("30sec", policy => {
    policy.Cache();
    policy.Expire(TimeSpan.FromSeconds(30));
});
...
```

The name given to a policy created with the `AddPolicy` method is used to apply the policy to an endpoint, like this:

```
...
app.MapEndpoint<Platform
    .SumEndpoint>("/sum30/{count:int=1000000000}")
    .CacheOutput("30sec");
...
```

The result is that requests to the `/sum` URL are cached using the default policy, while requests to the `/sum30` URL are cached using the `30sec` policy, causing the output from the endpoints to be cached for a different duration.

17.5 Using Entity Framework Core

Not all data values are produced directly by the application, and most projects will need to access data in a database. Entity Framework Core is well-integrated into the ASP.NET Core platform, with good support for creating a database from C# classes and for creating C# classes to represent an existing database. In the sections that follow, I demonstrate the process for creating a simple data model, using it to create a database, and querying that database in an endpoint.

> **Working with Entity Framework Core**
>
> The most common complaint about Entity Framework Core is poor performance. When I review projects that have Entity Framework Core performance issues, the problem is almost always because the development team has treated Entity Framework Core as a black box and has not paid attention to the SQL queries that are sent to the database. Not all LINQ features can be translated into SQL, and the most common problem is a query that retrieves large amounts of data from the database, which is then discarded after it has been reduced to produce a single value.
>
> Using Entity Framework Core requires a good understanding of SQL and ensuring that the LINQ queries made by the application are translated into efficient SQL queries. There are rare applications that have high-performance data requirements that cannot be met by Entity Framework Core, but that isn't the case for most typical web applications.
>
> That is not to say that Entity Framework Core is perfect. It has its quirks and requires an investment in time to become proficient. If you don't like the way that Entity Framework Core works, then you may prefer to use an alternative, such as Dapper (https://dapperlib.github.io/Dapper). But if your issue is that queries are not being performed fast enough, then you should spend some time exploring how those queries are being processed, which you can do using the techniques described in the remainder of the chapter.

17.5.1 Installing Entity Framework Core

Entity Framework Core requires a global tool package that is used to manage databases from the command line and to manage packages for the project that provides data access. To install the tools package, open a new PowerShell command prompt and run the commands shown in listing 17.20.

> **Listing 17.20 Installing the Entity Framework Core Global Tool Package**

```
dotnet tool uninstall --global dotnet-ef
dotnet tool install --global dotnet-ef --version 7.0.0
```

The first command removes any existing version of the `dotnet-ef` package, and the second command installs the version required for the examples in this book. This package provides the `dotnet ef` commands that you will see in later examples. To ensure the package is working as expected, run the command shown in listing 17.21.

Listing 17.21   Testing the Entity Framework Core Global Tool

```
dotnet ef --help
```

This command shows the help message for the global tool and produces the following output:

```
Entity Framework Core .NET Command-line Tools 7.0.0
Usage: dotnet ef [options] [command]
Options:
  --version       Show version information
  -h|--help       Show help information
  -v|--verbose    Show verbose output.
  --no-color      Don't colorize output.
  --prefix-output  Prefix output with level.
Commands:
  database    Commands to manage the database.
  dbcontext   Commands to manage DbContext types.
  migrations  Commands to manage migrations.
Use "dotnet ef [command] --help" for more information about a command.
```

Entity Framework Core also requires packages to be added to the project. If you are using Visual Studio Code or prefer working from the command line, navigate to the `Platform` project folder (the folder that contains the `Platform.csproj` file) and run the commands shown in listing 17.22.

Listing 17.22   Adding Entity Framework Core Packages to the Project

```
dotnet add package Microsoft.EntityFrameworkCore.Design --version 7.0.0
dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 7.0.0
```

17.5.2   *Creating the data model*

For this chapter, I am going to define the data model using C# classes and use Entity Framework Core to create the database and schema. Create the `Platform/Models` folder and add to it a class file called `Calculation.cs` with the contents shown in listing 17.23.

Listing 17.23   The contents of the Calculation.cs file in the Platform/Models folder

```
namespace Platform.Models {

    public class Calculation {
        public long Id { get; set; }
        public int Count { get; set; }
        public long Result { get; set; }
    }
}
```

You can see more complex data models in other chapters, but for this example, I am going to keep with the theme of this chapter and model the calculation performed in earlier examples. The `Id` property will be used to create a unique key for each object stored in the database, and the `Count` and `Result` properties will describe a calculation and its result.

Entity Framework Core uses a context class that provides access to the database. Add a file called `CalculationContext.cs` to the `Platform/Models` folder with the content shown in listing 17.24.

Listing 17.24   The CalculationContext.cs file in the Platform/Models folder

```
using Microsoft.EntityFrameworkCore;

namespace Platform.Models {

    public class CalculationContext : DbContext {

        public CalculationContext(
            DbContextOptions<CalculationContext> opts) : base(opts) { }

        public DbSet<Calculation> Calculations => Set<Calculation>();
    }
}
```

The `CalculationContext` class defines a constructor that is used to receive an options object that is passed on to the base constructor. The `Calculations` property provides access to the `Calculation` objects that Entity Framework Core will retrieve from the database.

17.5.3 Configuring the database service

Access to the database is provided through a service, as shown in listing 17.25.

Listing 17.25   Configuring the data service in the Program.cs file in the Platform folder

```
using Platform.Services;
using Platform.Models;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddOutputCache(opts => {
    opts.AddBasePolicy(policy => {
        policy.Cache();
        policy.Expire(TimeSpan.FromSeconds(10));
    });
    opts.AddPolicy("30sec", policy => {
        policy.Cache();
        policy.Expire(TimeSpan.FromSeconds(30));
    });
});

builder.Services.AddSingleton<IResponseFormatter,
    HtmlResponseFormatter>();

builder.Services.AddDbContext<CalculationContext>(opts => {
    opts.UseSqlServer(
        builder.Configuration["ConnectionStrings:CalcConnection"]);
});
```

```
var app = builder.Build();

app.UseOutputCache();

app.MapEndpoint<Platform
    .SumEndpoint>("/sum/{count:int=1000000000}")
    .CacheOutput();

app.MapEndpoint<Platform
    .SumEndpoint>("/sum30/{count:int=1000000000}")
    .CacheOutput("30sec");

app.MapGet("/", async context => {
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

The `AddDbContext` method creates a service for an Entity Framework Core context class. The method receives an options object that is used to select the database provider, which is done with the `UseSqlServer` method. The `IConfiguration` service is used to get the connection string for the database, which is defined in listing 17.26.

> ### Listing 17.26   A connection string in the appsettings.json file in the Platform folder

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.EntityFrameworkCore": "Information"
    }
  },
  "AllowedHosts": "*",
  "Location": {
    "CityName": "Buffalo"
  },
  "ConnectionStrings": {
    "CacheConnection": "Server=(localdb)\\MSSQLLocalDB;Database=CacheDb",
    "CalcConnection": "Server=(localdb)\\MSSQLLocalDB;Database=CalcDb"
  }
}
```

The listing also sets the logging level for the `Microsoft.EntityFrameworkCore` category, which will show the SQL statements that are used by Entity Framework Core to query the database.

> **TIP** Set the `MultipleActiveResultSets` option to `True` for connection strings that will be used to make queries with multiple result sets. You can see an example of this option set in the connection strings for the SportsStore project in chapter 7.

17.5.4 *Creating and applying the database migration*

Entity Framework Core manages the relationship between data model classes and the database using a feature called *migrations*. When changes are made to the model classes, a new migration is created that modifies the database to match those changes. To create the initial migration, which will create a new database and prepare it to store `Calculation` objects, open a new PowerShell command prompt, navigate to the folder that contains the `Platform.csproj` file, and run the command shown in listing 17.27.

> **Listing 17.27   Creating a migration**

```
dotnet ef migrations add Initial
```

The `dotnet ef` commands relate to Entity Framework Core. The command in listing 17.27 creates a new migration named `Initial`, which is the name conventionally given to the first migration for a project. You will see that a `Migrations` folder has been added to the project and that it contains class files whose statements prepare the database so that it can store the objects in the data model. To apply the migration, run the command shown in listing 17.28 in the `Platform` project folder.

> **Listing 17.28   Applying a migration**

```
dotnet ef database update
```

This command executes the commands in the migration and uses them to prepare the database, which you can see in the SQL statements written to the command prompt.

17.5.5 *Seeding the database*

Most applications require some seed data, especially during development. Entity Framework Core does provide a database seeding feature, but it is of limited use for most projects because it doesn't allow data to be seeded where the database allocates unique keys to the objects it stores. This is an important feature in most data models because it means the application doesn't have to worry about allocating unique key values.

A more flexible approach is to use the regular Entity Framework Core features to add seed data to the database. Create a file called `SeedData.cs` in the `Platform/Models` folder with the code shown in listing 17.29.

> **Listing 17.29   The contents of the SeedData.cs file in the Platform/Models folder**

```
using Microsoft.EntityFrameworkCore;

namespace Platform.Models {
    public class SeedData {
        private CalculationContext context;
        private ILogger<SeedData> logger;

        private static Dictionary<int, long> data
            = new Dictionary<int, long>() {
```

```
                    {1, 1}, {2, 3}, {3, 6}, {4, 10}, {5, 15},
                    {6, 21}, {7, 28}, {8, 36}, {9, 45}, {10, 55}
            };

        public SeedData(CalculationContext dataContext,
                ILogger<SeedData> log) {
            context = dataContext;
            logger = log;
        }

        public void SeedDatabase() {
            context.Database.Migrate();
            if (context.Calculations?.Count() == 0) {
                logger.LogInformation("Preparing to seed database");
                context.Calculations.AddRange(
                        data.Select(kvp => new Calculation() {
                            Count = kvp.Key, Result = kvp.Value
                        }));
                context.SaveChanges();
                logger.LogInformation("Database seeded");
            } else {
                logger.LogInformation("Database not seeded");
            }
        }
    }
}
```

The `SeedData` class declares constructor dependencies on the `CalculationContext` and `ILogger<T>` types, which are used in the `SeedDatabase` method to prepare the database. The context's `Database.Migrate` method is used to apply any pending migrations to the database, and the `Calculations` property is used to store new data using the `AddRange` method, which accepts a sequence of `Calculation` objects.

The new objects are stored in the database using the `SaveChanges` method. To use the `SeedData` class, make the changes shown in listing 17.30 to the `Program.cs` file.

> **Listing 17.30   Enabling database seeding in the Program.cs file in the Platform folder**

```
using Microsoft.EntityFrameworkCore;
using Platform.Models;
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddOutputCache(opts => {
    opts.AddBasePolicy(policy => {
        policy.Cache();
        policy.Expire(TimeSpan.FromSeconds(10));
    });
    opts.AddPolicy("30sec", policy => {
        policy.Cache();
        policy.Expire(TimeSpan.FromSeconds(30));
    });
});
```

```
builder.Services.AddSingleton<IResponseFormatter,
    HtmlResponseFormatter>();

builder.Services.AddDbContext<CalculationContext>(opts => {
    opts.UseSqlServer(
        builder.Configuration["ConnectionStrings:CalcConnection"]);
});

builder.Services.AddTransient<SeedData>();

var app = builder.Build();

app.UseOutputCache();

app.MapEndpoint<Platform
    .SumEndpoint>("/sum/{count:int=1000000000}")
    .CacheOutput();

app.MapEndpoint<Platform
    .SumEndpoint>("/sum30/{count:int=1000000000}")
    .CacheOutput("30sec");

app.MapGet("/", async context => {
    await context.Response.WriteAsync("Hello World!");
});

bool cmdLineInit = (app.Configuration["INITDB"] ?? "false") == "true";
if (app.Environment.IsDevelopment() || cmdLineInit) {
    var seedData = app.Services.GetRequiredService<SeedData>();
    seedData.SeedDatabase();
}
if (!cmdLineInit) {
    app.Run();
}
```

I create a service for the `SeedData` class, which means that it will be instantiated, and
its dependencies will be resolved, which is more convenient than working directly with
the class constructor.

   If the hosting environment is `Development`, the database will be seeded automati-
cally as the application starts. It can also be useful to seed the database explicitly, espe-
cially when setting up the application for staging or production testing. This statement
checks for a configuration setting named `INITDB`:

```
...
bool cmdLineInit = (app.Configuration["INITDB"] ?? "false") == "true";
...
```

This setting can be supplied on the command line to seed the database, after which the
application will terminate because the `Run` method, which starts listening for HTTP
requests, is never called.

   To seed the database, open a new PowerShell command prompt, navigate to the
project folder, and run the command shown in listing 17.31.

> **Listing 17.31 Seeding the database**

```
dotnet run INITDB=true
```

The application will start, and the database will be seeded with the results for the ten calculations defined by the SeedData class, after which the application will terminate. During the seeding process, you will see the SQL statements that are sent to the database, which check to see whether there are any pending migrations, count the number of rows in the table used to store Calculation data, and, if the table is empty, add the seed data.

> **NOTE** If you need to reset the database, you can use the dotnet ef database drop --force command. You can then use dotnet run INITDB=true to re-create and seed the database again.

17.5.6 *Using data in an endpoint*

Endpoints and middleware components access Entity Framework Core data by declaring a dependency on the context class and using its DbSet<T> properties to perform LINQ queries. The LINQ queries are translated into SQL and sent to the database. The row data received from the database is used to create data model objects that are used to produce responses. Listing 17.32 updates the SumEndpoint class to use Entity Framework Core.

> **Listing 17.32 Using a database in the SumEndpoint.cs file in the Platform folder**

```
//using Platform.Services;
using Platform.Models;

namespace Platform {

    public class SumEndpoint {

        public async Task Endpoint(HttpContext context,
                CalculationContext dataContext) {

            int count;
            int.TryParse((string?)context.Request.RouteValues["count"],
                out count);

            long total = dataContext.Calculations?
                .FirstOrDefault(c => c.Count == count)?.Result ?? 0;
            if (total == 0) {
                for (int i = 1; i <= count; i++) {
                    total += i;
                }
                dataContext.Calculations?.Add(new() {
                    Count = count, Result = total
                });
                await dataContext.SaveChangesAsync();
            }
```

```
            string totalString = $"({DateTime.Now.ToLongTimeString()}) "
                + total;

            await context.Response.WriteAsync(
                $"({DateTime.Now.ToLongTimeString()}) Total for {count}"
                + $" values:\n{totalString}\n");
        }
    }
}
```

The endpoint uses the LINQ `FirstOrDefault` to search for a stored `Calculation` object for the calculation that has been requested like this:

```
...
dataContext.Calculations?
    .FirstOrDefault(c => c.Count == count)?.Result ?? 0;
...
```

If an object has been stored, it is used to prepare the response. If not, then the calculation is performed, and a new `Calculation` object is stored by these statements:

```
...
dataContext.Calculations?.Add(new () { Count = count, Result = total});
await dataContext.SaveChangesAsync();
...
```

The `Add` method is used to tell Entity Framework Core that the object should be stored, but the update isn't performed until the `SaveChangesAsync` method is called. To see the effect of the changes, restart ASP.NET Core MVC (without the `INITDB` argument if you are using the command line) and request the http://localhost:5000/sum/10 URL. This is one of the calculations with which the database has been seeded, and you will be able to see the query sent to the database in the logging messages produced by the application.

```
...
Executing DbCommand [Parameters=[@__count_0='?' (DbType = Int32)],
    CommandType='Text', CommandTimeout='30']
SELECT TOP(1) [c].[Id], [c].[Count], [c].[Result]
      FROM [Calculations] AS [c]
      WHERE [c].[Count] = @__count_0
...
```

If you request http://localhost:5000/sum/100, the database will be queried, but no result will be found. The endpoint performs the calculation and stores the result in the database before producing the result shown in figure 17.6.
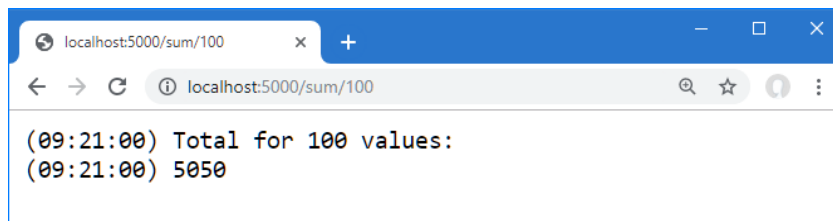

Figure 17.6 Performing a calculation

Once a result has been stored in the database, subsequent requests for the same URL will be satisfied using the stored data. You can see the SQL statement used to store the data in the logging output produced by Entity Framework Core.

```
...
Executing DbCommand [Parameters=[@p0='?' (DbType = Int32),
    @p1='?' (DbType = Int64)],
        CommandType='Text', CommandTimeout='30']
SET NOCOUNT ON;
INSERT INTO [Calculations] ([Count], [Result])
VALUES (@p0, @p1);
SELECT [Id]
FROM [Calculations]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
...
```

> **NOTE**   Notice that the data retrieved from the database is not cached and that each request leads to a new SQL query. Depending on the frequency and complexity of the queries you require, you may want to cache data values or responses using the techniques described earlier in the chapter.

ENABLING SENSITIVE DATA LOGGING

Entity Framework Core doesn't include parameter values in the logging messages it produces, which is why the logging output contains question marks, like this:

```
...
Executing DbCommand [Parameters=[@__count_0='?' (DbType = Int32)],
    CommandType='Text', CommandTimeout='30']
...
```

The data is omitted as a precaution to prevent sensitive data from being stored in logs. If you are having problems with queries and need to see the values sent to the database, then you can use the `EnableSensitiveDataLogging` method when configuring the database context, as shown in listing 17.33.

> **Listing 17.33**   **Sensitive data logging in the Program.cs file in the Platform folder**

```
...
builder.Services.AddDbContext<CalculationContext>(opts => {
    opts.UseSqlServer(
        builder.Configuration["ConnectionStrings:CalcConnection"]);
    opts.EnableSensitiveDataLogging(true);
});
...
```

Restart ASP.NET Core MVC and request the http://localhost:5000/sum/100 URL again. When the request is handled, Entity Framework Core will include parameter values in the logging message it creates to show the SQL query, like this:

```
...
Executed DbCommand (40ms) [Parameters=[@__count_0='100'],
    CommandType='Text', CommandTimeout='30']
SELECT TOP(1) [c].[Id], [c].[Count], [c].[Result]
```

```
FROM [Calculations] AS [c]
WHERE [c].[Count] = @__count_0
...
```

This is a feature that should be used with caution because logs are often accessible by people who would not usually have access to the sensitive data that applications handle, such as credit card numbers and account details.

Summary

- ASP.NET Core provides support for caching individual data values, which can be accessed by endpoints.
- Entire responses can be cached, which means that requests are serviced from the cache without using the endpoint.
- ASP.NET Core provides two different middleware components for caching responses. The response cache requires a header to be set and only uses cached responses for some requests. The output cache is more complex, but more comprehensive, and uses cached responses more widely.
- Entity Framework Core provides access to relational data. Entity Framework Core is configured as a service and consumed by endpoints via dependency injection. Entity Framework Core can be used to create a database from a set of model classes and is queried using LINQ.